Esta clase va a ser

- grabada

Clase 23. PROGRAMACIÓN BACKEND

Ruteo avanzado y manejo de políticas de autorización

CODERHOUSE

Temario

22

Passport Avanzado

- ✓ Formas de envío de JWT
- ✓ Estrategia de JWT
- ✓ Control interno de mensajes y sistema

23

Ruteo Avanzado y manejo de políticas de autorización

- ✓ Estrategias avanzadas de Router
- ✓ Custom Router
- ✓ Aprovechando al máximo Custom Router

24

Segunda práctica integradora

- ✓ Skills
- ✓ Práctica integradora

CODERHOUSE

Objetivos de la clase

- Repasar conceptos de ruteo en Express
- Crear un router personalizado
- Manejar las políticas de auth en este router personalizado

CODERHOUSE

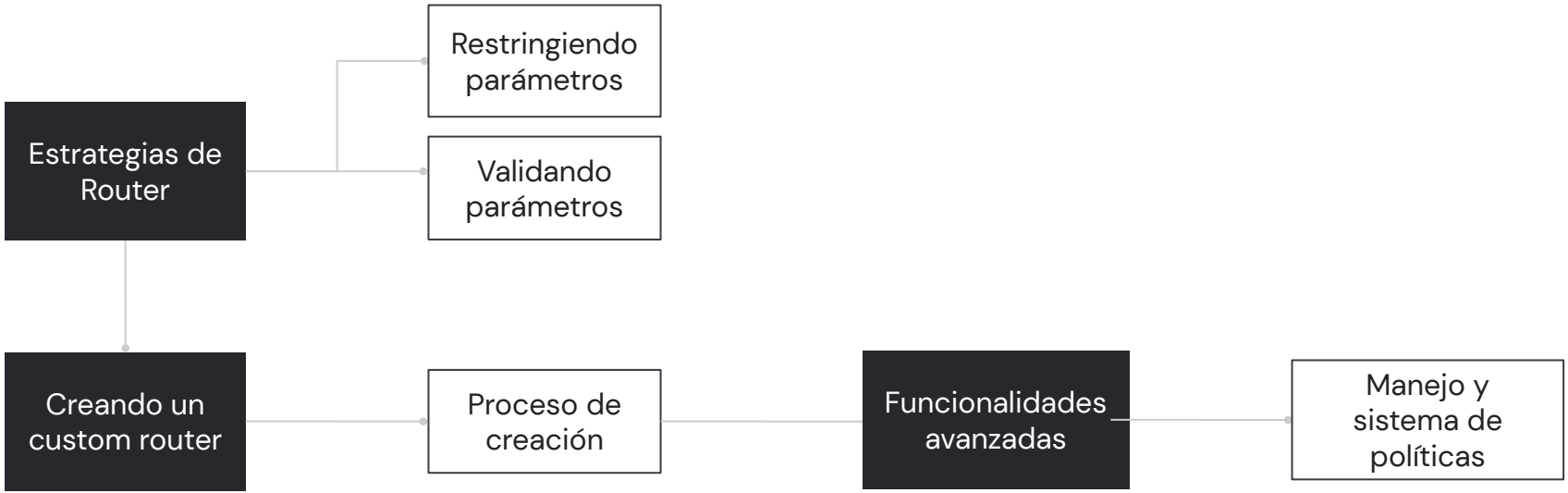
CLASE Nº 22

Glosario

JWT: Estándar que define una manera segura de transmitir información entre dos partes como objeto de tipo JSON. Generalmente se utiliza para crear tokens de autenticación con el fin de delegar la responsabilidad de sesión al frontend.

localStorage: Es un recurso que persiste información en un almacenamiento interno del navegador.

MAPA DE CONCEPTOS



Estrategias avanzadas de Router

Mejorando las prácticas de router

A estas alturas has podido relacionarte ya con el router en cuanto a:

- ✓ Definición de rutas
- ✓ Utilización de middlewares
- ✓ Callbacks para manejo de peticiones y respuestas

Sin embargo, conforme el proyecto crece, notaremos que algunos conceptos son aplicados de una manera un tanto diferente a lo visto en clases previas.

En esta clase abordaremos algunos conceptos y técnicas necesarias para profesionalizar el uso de router y poder encontrar soluciones óptimas para problemas del mundo real.

Restringiendo parámetros

No todos los parámetros son buenos

Imagina que tienes una ruta de la siguiente forma:

```
app.get("/api/dictionary/:word", (req, res) => {
  // Procesamiento de la request
});
```

Esto indica que la ruta debe recibir una palabra. El usuario entonces puede colocar:

- ✓ Una palabra en mayúsculas: “PERRO”
- ✓ Una palabra en minúsculas : “gato”

Pero también puede colocar:

- ✓ Un número : “12342322”.
- ✓ Una cadena mixta: “3$adac331”

Por lo tanto, en muchos casos nos será conveniente **validar** que la información enviada sea correcta..

¿Cómo recibir sólo lo esperado?

Existen múltiples técnicas para poder abordar esta situación. Primero, bajo este ejemplo, podemos separar la lógica de nuestro dictionary en un router propio. Y posteriormente aplicar una **expresión regular** que permita tomar solo la ruta indicada.

JS dictionary.router.js

src > routes > JS dictionary.router.js > ...

```
2
3   const dictionaryRouter = new Router();
4
5   // La expresión regular [a-zA-Z]+ indica que el parámetro debe ser un valor alfabético
6   dictionaryRouter.get("/:word([a-zA-Z]+)", (req, res) => {
7     res.send(req.params.word);
8   });
9
10  export default dictionaryRouter;
```

CODERHOUSE

¿Qué conseguimos con esto?

Aplicar una expresión regular en un parámetro, indica que dicho parámetro será aceptado **sólo si se cumple la expresión regular indicada en paréntesis**.

En caso de que el parámetro cumpla correctamente con la expresión regular, la petición se realizará normalmente/

Con esta expresión regular acabamos de controlar:

- ✓ Parámetro vacío
- ✓ Parámetros numéricos
- ✓ Parámetros con caracteres especiales
- ✓ Parámetros mixtos

¿Si necesitáramos caracteres diferentes?

Cómo hispano hablantes, sabemos que los caracteres como á, é, í, ó, ú, ü, etc, son de uso común. Por lo tanto, excluir de un diccionario todas las palabras que cuenten con estos caracteres no sería correcto. Para poder manejar esto desde la expresión regular, hay que agregar los valores que deseamos que también sean considerados. **Esto debe agregarse a la expresión regular utilizando los valores url-encoded de los caracteres:**

```
dictionaryRouter.get(
  "/:word([a-zA-Z%C3%A1%C3%A9%C3%AD%C3%B3%C3%BA%C3%BC]+)",
  (req, res) => {
    res.send(req.params.word);
  }
);
```

Donde:

á = %C3%A1
é = %C3%A9
í = %C3%AD
ó = %C3%B3
ú = %C3%BA
ü = %C3%BC

¿Qué hacer con todas las rutas que no coinciden con ningún endpoint?

Dado que estamos trabajando a nivel router, para los casos donde la palabra no cumpla con la expresión regular indicada, podemos indicar un get * a nivel router para indicar que no se está cumpliendo con la ruta.

```
dictionaryRouter.get("*", (req, res) =>
  res.status(404).send("Cannot get the specified word")
);
```

Notar que, si definimos esta ruta luego de la original, la misma va a 'atrapar' los casos que la original omitió (en función de sus normas de validación).

Validando parámetros

CODERHOUSE

Supongamos que nuestro router crece

Al crecer, incorporamos múltiples endpoints que sirven para poder trabajar con las palabras del diccionario.

En este caso, en cada uno de ellos requerimos el parámetro **word.**

```
dictionaryRouter.get(
  "/:word([a-zA-Z%C3%A1%C3%A9%C3%AD%C3%B3%C3%BA%C3%BC]+)",
  (req, res) => {
    res.send(req.params.word);
  }
);

dictionaryRouter.get(
  "/:word([a-zA-Z%C3%A1%C3%A9%C3%AD%C3%B3%C3%BA%C3%BC]+)/:language([a-z])",
  (req, res) => {
    //
  }
);

dictionaryRouter.put(
  "/:word([a-zA-Z%C3%A1%C3%A9%C3%AD%C3%B3%C3%BA%C3%BC]+)/:language([a-z])",
  (req, res) => {
    //
  }
);

dictionaryRouter.delete(
  "/:word([a-zA-Z%C3%A1%C3%A9%C3%AD%C3%B3%C3%BA%C3%BC]+)/:language([a-z])",
  (req, res) => {
    //
  }
);
```

Hay una operación en común

Para cualquiera de los endpoints mostrados anteriormente, necesitaremos hacer lo siguiente:

- ✓ Obtener el parámetro **word**.
- ✓ Buscar el parámetro en la base de datos para validar que se encuentre dentro de nuestro sistema de persistencia.
- ✓ Una vez validado, continuar con la operación que queremos.

A sabiendas de que ésto puede repetirse una y otra vez según los endpoints que vayan utilizando el parámetro word, existe una forma de regular el uso de éste: **router.param**

router.param

router.param funciona como un middleware **específicamente para el parámetro matcheado**. Este se manda a llamar después de los middlewares principales y tiene como función el generalizar las operaciones hechas con dicho parámetro. La sintaxis es:

```
dictionaryRouter.param(
  "nombre del parametro",
  async (req, res, next, valorDelParametro) => {
    //
  }
);
```

En nuestro caso particular de un parámetro word, la función será:

```
dictionaryRouter.param("word", async (req, res, next, word) => {
  //
});
```

Ejemplo de uso de router.param para encontrar una palabra

```
dictionaryRouter.param("word", async (req, res, next, word) => {
  const searchedWord = await dictionaryService.findWord(word);

  if (!searchedWord) {
    req.word = null;
  } else {
    req.word = searchedWord;
  }

  next();
});
```

De esta manera, `req.word` aparecerá en todos los endpoints que utilicen el parámetro :word

Router de mascotas

Individual
Duración: 15 min

CODERHOUSE

ACTIVIDAD EN CLASE

Router de mascotas

Desarrollar un servidor sencillo utilizando Express

Crear un router para manejo de mascotas en una ruta base '/api/pets'. Este gestionará diferentes mascotas utilizando como persistencia un array local. Posteriormente, desarrollar los siguientes endpoints:

- ✓ POST ('/'): deberá insertar una nueva mascota. El formato de cada mascota será `{name:String, species: String}`
- ✓ GET('/:pet'): Deberá traer la mascota con el nombre indicado. Utilizar una expresión regular para que sólo se puedan recibir letras e incluso espacios (recuerda cómo se lee un espacio a nivel URL). No debe permitir números.
- ✓ PUT ('/:pet'): Deberá traer la mascota y añadirle un campo "adopted:true" a dicha mascota en caso de existir.
- ✓ Generar además un router.param que permita acceder de manera directa a la mascota, colocándola en **req.pet**

Creando un custom router

El router no es tan simple como parece

Hemos realizado innumerables veces el router de Express para poder conectar los endpoints que necesitamos, sin embargo, hay algunas limitantes en la escalabilidad y limpieza de nuestro código con las implementaciones de router que se tienen actualmente.

¿Por qué customizar un router que ya existe?

Conforme nuestra aplicación crece y el equipo de desarrollo crece, es importante que las implementaciones que hagamos sean lo más normalizadas posibles, así también como lo más ajustado a las necesidades de nuestra empresa.

Crearemos nuestro propio router a partir del router de Express.

Ventajas de un custom Router

- ✓ Manejo sistematizado de respuestas: Si un desarrollador nuevo llegara a tu equipo, ¿cómo explicas el formato específico que tiene que responder cada operación?

  Customizar el router permitirá crear “respuestas predefinidas” en el objeto **res**, con el fin de que el equipo pueda utilizarlo sin preocupación de utilizar mal algún formato.

  Podemos abstraer el `res.send()` habitual y convertirlo en respuestas más intuitivas como `res.sendSuccessMessage`, `res.sendError`, `res.sendSuccessPayload`, etc.

Ventajas de un custom Router

- ✓ Gestión de middlewares interiorizada: Cuando hacemos un app.use(middleware), estamos colocando un middleware para todos nuestros endpoints.

  ¿Pero si necesitamos que cada endpoint tenga un input dinámico para procesar dicho middleware? Es entonces cuando tenemos que crear middlewares a nivel router, y llenar nuestros endpoints con middlewares escritos explícitamente para colocar sus inputs.

  Customizar el router permitirá dinamizar el middleware generalizado como si fuera un parámetro más del endpoint.

Proceso de creación

APROXIMACIÓN AL PROCESO

Primero, definiremos en nuestra carpeta routes una clase app.router.js. Ésta contendrá la clase principal de todo nuestro router.

Asimismo, nuestro AppRouter tendrá como propiedad interna un Router de Express.

app.router.js

src > routes > app.router.js > ...

```
1  import { Router } from "express";
2
3  export default class AppRouter {
4    #router;
5
6    constructor() {
7      this.#router = Router();
8      this.init();
9    }
10
11   get router() {
12     return this.#router;
13   }
14
15   init() {
16     // Para ser implementado por las clases hijas
17   }
18
19   get(path, ...callbacks) {
20     this.router.get(path, this.#applyCallbacks(callbacks));
21   }
```

APROXIMACIÓN AL PROCESO

Ahora, necesitamos una función que procese todas las funciones internas del router (middlewares y el request handler). La llamaremos “applyCallbacks”

```
/**
 * Esta función ejecutará los callbacks de uno a uno
 */
#applyCallbacks(callbacks) {
  return callbacks.map((cb) => async (...params) => {
    try {
      /**
       * Aquí optamos, en lugar de ejecutar el callback directamente, por ejecutarlo utilizando apply,
       * lo cual nos permitirá ejecutarlos con el contexto de la instancia de router que los ejecute.
       * En cuanto a los parametros, los mismos seran (req, res, next), correspondientes a
       * middlewares/request handlers de Express.
       */
      await cb.apply(this, params);
    } catch (err) {
      console.log(err);

      // Params[1] se corresponderá siempre con el objecto res, por lo cual en caso de error
      // puedo enviar el error con el método send y un status 500
      params[1].status(500).send(err);
    }
  });
}
```

APROXIMACIÓN AL PROCESO

Para poder crear un router específico, lo crearemos como una clase también, ésta heredada de nuestro "AppRouter" genérico.

JS users.router.js ×

src > routes > JS users.router.js > ...

```
1   import AppRouter from "./app.router.js";
2
3   export default class UsersRouter extends AppRouter {
4     // Hacemos uso de init para inicializar nuestras rutas.
5     // En este caso, esto sería equivalente a ejecutar router.get
6     init() {
7       this.get("/", (req, res) => {
8         res.send("Hola, coders!");
9       });
10    }
11  }
```

CODERHOUSE

APROXIMACIÓN AL PROCESO

¿Cómo conectamos app con este "nuevo" Router? utilizaremos el middleware de router habitual, sólo que debemos antes instanciar el Router, y utilizar el atributo .router del mismo para obtener el router de Express.

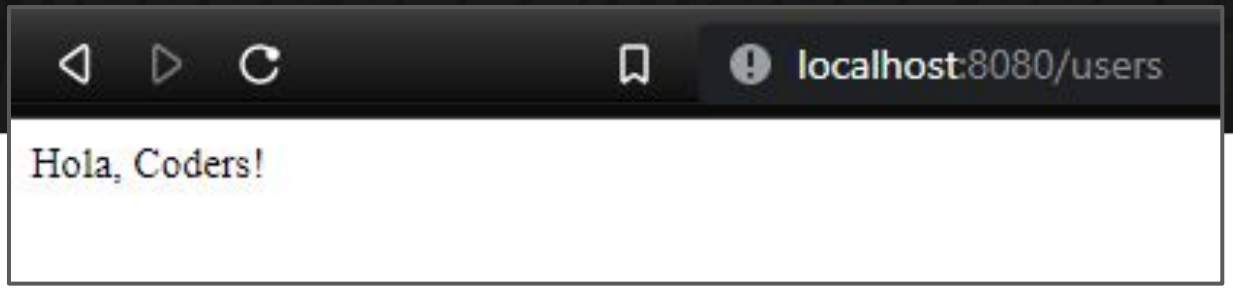
JS app.js

src > JS app.js > ...

```
1  import express from "express";
2  import UsersRouter from "./routes/users.router.js";
3
4  const app = express();
5
6  const usersRouter = new UsersRouter();
7
8  app.use("/users", usersRouter.router);
9
10 app.listen(8080, () => {
11   console.log("Server listening on port 8080");
12 });
```

APROXIMACIÓN AL PROCESO

Si hacemos todo correctamente, podremos inicializar nuestro servidor como siempre lo hemos hecho, al visitar la ruta, el router se encargará de todo.



Para pensar

¿Y ahora? La estructura de un router customizado no parece ser la gran cosa mostrada así.
¿Qué utilidad adicional podríamos sumar a esto?

¡Tomate un momento para pensarlo!

Break

¡10 minutos y volvemos!

Aprovechando al máximo el Custom Router

Custom responses

Una manera bastante limpia y escalable de trabajar en un equipo de backend, es que todas las respuestas mantengan un formato específico para que el frontend no tenga que trabajar con diferentes respuestas de diferentes formatos.

Setear una custom response implica dejar respuestas predefinidas con el fin de que los desarrolladores sólo tengan que llamarlas, llenando los datos solicitados.

Utilizaremos un middleware interno del router para poder crear y ligar dichas respuestas, **el cual ligaremos no solo a una ruta, sino al método get, post, put y delete propios del router**

addCustomResponses

`addCustomResponses` es una función que agregará al objeto **res**, métodos adicionales de envío de información, donde seteamos status específicos, cuerpos específicos e intenciones específicas.

```
#addCustomResponses(req, res, next) {
  /**
   * Estos métodos permiten que solo tengamos que enviar el payload,
   * y el status se setee automáticamente.
   */
  res.sendSuccess = (payload) => res.send({ status: "success", payload });
  res.sendServerError = (error) =>
    res.status(500).send({ status: "error", error });
  res.sendClientError = (error) =>
    res.status(400).send({ status: "error", error });

  next();
}
```

Conectando las nuevas respuestas a nuestros métodos principales

Al final, nuestros métodos principales de router quedarán homologados con las respuestas correspondientes y podremos utilizar en nuestro router la respuesta directamente.

```
get(path, ...callbacks) {
  this.router.get(path, this.#addCustomResponses, this.#applyCallbacks(callbacks));
}

post(path, ...callbacks) {
  this.router.post(path, this.#addCustomResponses, this.#applyCallbacks(callbacks));
}

put(path, ...callbacks) {
  this.router.put(path, this.#addCustomResponses, this.#applyCallbacks(callbacks));
}

delete(path, ...callbacks) {
  this.router.delete(path, this.#addCustomResponses, this.#applyCallbacks(callbacks));
}
```

JS users.router.js

src > routes > JS users.router.js > ...

```
1  import AppRouter from "./app.router.js";
2
3  export default class UsersRouter extends AppRouter {
4    init() {
5      this.get("/", (req, res) => {
6        res.sendSuccess("Hola, coders!");
7      });
8    }
9  }
```

CODERHOUSE

Manejo de políticas

CODERHOUSE

La autorización a nivel escalable

Estamos acostumbrados a validar, como método de autorización, que el usuario sea administrador o sea usuario. Es decir, la validación habitual que realizamos sólo sirve para definir dos roles.

Sin embargo, muy difícilmente encontramos un backend que sólo se base en esos dos roles; generalmente tendremos la necesidad de definir una mayor cantidad de roles, así como también una mayor granularidad en el control de recursos en base a dichos roles.

La autorización a nivel escalable

Imagina que en tu trabajo estás desarrollando un ecommerce, y te comentan que requieren rutas validadas a diferentes niveles. Por ejemplo:

- ✓ **Endpoints públicos**. Es decir, puede acceder a este servicio cualquier cliente.
- ✓ **Endpoints autenticados**. Es decir, puede acceder a este servicio cualquier cliente que haya sido previamente autenticado.
- ✓ **Endpoints para usuarios**. Puede acceder cualquier cliente con dicho rol.
- ✓ **Endpoints para usuarios premium**. Puede acceder cualquier cliente con rol de usuario premium
- ✓ **Endpoints para administradores**. Puede acceder solo el o los administradores del sitio
- ✓ **Endpoints mixtos**. Por ejemplo, combinaciones de usuarios y usuarios premium, administradores y usuarios premium, etc.

Para estos casos, el modelo a utilizar deberá ser algo más complejo.

Los sistemas de políticas varían según la empresa

Cada empresa tiene sus requisitos a nivel seguridad y como desarrolladores debemos estar abiertos a cualquier combinación de permisos que la empresa requiera. En este curso abordaremos las siguientes políticas:

- ✓ `PUBLIC` : acceso libre
- ✓ `AUTHENTICATED`: que cuente con token de acceso (comprende a usuarios, premiums y admins)
- ✓ `USER`: usuario común
- ✓ `USER_PREMIUM`: usuario con membresía premium
- ✓ `ADMIN`: Administrador del sitio.

Hands on lab

En esta instancia de la clase **conectaremos** un sistema de política, utilizando jwt y nuestro custom Router.

¿De qué manera?

El profesor demostrará cómo hacerlo y tú lo puedes ir replicando en tu computadora. Si surgen dudas las puedes compartir para resolverlas en conjunto de la mano de los tutores.

Tiempo estimado: **25-35 minutos**

CODERHOUSE

HANDS ON LAB

Sistema de políticas

¿Cómo lo hacemos? **Se creará un middleware que, en cada momento, valide el acceso a partir de las políticas (Sólo PUBLIC, USER, ADMIN).**

- ✓ Definir una función **handlePolicies**,
- ✓ El método deberá recibir un **arreglo de políticas (strings)** y seguir los siguientes pasos:
  - Si la única política es “PUBLIC”, continuar sin problema.
  - para el resto de casos, primero procesar el token de jwt, el cual llegará desde los headers de autorización.
  - Posteriormente, validar el rol del usuario que esté dentro del token para corroborar si se encuentra dentro de las políticas
- ✓ Cada método get, post, put, delete deberá recibir **antes de los callbacks**, un arreglo de políticas.
- ✓ Colocar handlePolicies como el primer middleware del procesamiento.

HANDS ON LAB

Sistema de políticas

Una vez configurado el router:

- ✓ Crear un router **session** que cuente con un endpoint '/login', el cual guardará al usuario en un token (no es necesario un registro, todo es hardcodeado para agilizar el proceso de políticas)
- ✓ En el router de usuario que ya existe, crear un endpoint que utilice las políticas de usuario, y otro que utilice sólo políticas públicas.
- ✓ Validar políticas con Postman.

CONSIDERACIONES:

1. Recordar agregar app.use(express.json()); para el login
2. Recordar agregar a los headers de autorización el Bearer token en Postman.
3. Enviar el jwt por medio de body para este caso
4. Hardcodear el rol del cliente en "user".

#handlePolicies

```
#handlePolicies(policies) {
  return (req, res, next) => {
    if (policies.includes("PUBLIC")) {
      return next();
    }

    const authHeaders = req.headers.authorization;
    if (!authHeaders) {
      return res.status(401).send({ status: "error", error: "Unauthorized" });
    }

    const token = authHeaders.split(" ")[1]; // Removemos el "Bearer "
    const user = jwt.verify(token, "CoderSecret"); // Obtenemos el usuario a partir del jwt

    // Verificamos si el rol del usuario está dentro de las políticas permitidas para este endpoint
    if (!policies.includes(user.role.toUpperCase())) {
      return res.status(403).send({ status: "error", error: "Forbidden" });
    }

    req.user = user;
    next();
  };
}
```

Aplicando #handlePolicies a los métodos get, post, put, delete

```
get(path, policies, ...callbacks) {
  this.router.get(path, this.#handlePolicies(policies), this.#addCustomResponses, this.#applyCallbacks(callbacks));
}

post(path, policies, ...callbacks) {
  this.router.post(path, this.#handlePolicies(policies), this.#addCustomResponses, this.#applyCallbacks(callbacks));
}

put(path, policies, ...callbacks) {
  this.router.put(path, this.#handlePolicies(policies), this.#addCustomResponses, this.#applyCallbacks(callbacks));
}

delete(path, policies, ...callbacks) {
  this.router.delete(path, this.#handlePolicies(policies), this.#addCustomResponses, this.#applyCallbacks(callbacks));
}
```

Sessions Router con login hardCodeado (No estamos aquí para validar cosas con el login, sino para practicar políticas).

```
import AppRouter from "./app.router.js";
import jwt from "jsonwebtoken";

export default class SessionsRouter extends AppRouter {
  init() {
    this.post("/login", ["PUBLIC"], (req, res) => {
      const user = {
        email: req.body.email,
        role: "user",
      };

      const token = jwt.sign(user, "CoderSecret");

      res.sendSuccess({ token });
    });
  }
}
```

UsersRouter, ahora todas las rutas tienen un parámetro intermedio con todos los roles que serán aceptados (recuerda que PUBLIC es para todos)

```
import AppRouter from "./app.router.js";

export default class UsersRouter extends AppRouter {
  init() {
    this.get("/", ["PUBLIC"], (req, res) => {
      res.sendSuccess("Hola, coders!");
    });

    this.get("/me", ["USER", "USER PREMIUM", "ADMIN"], (req, res) => {
      res.sendSuccess(req.user);
    });
  }
}
```

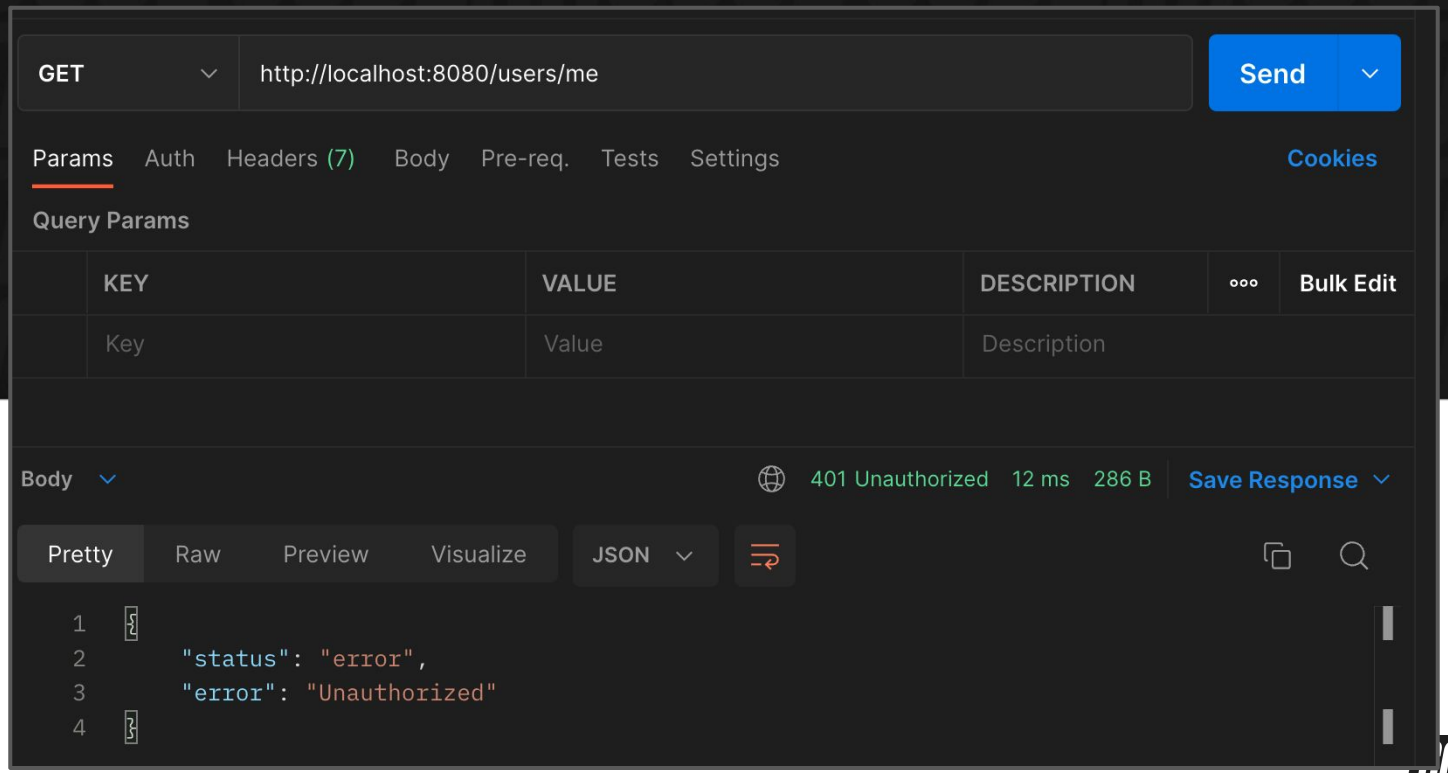
Configurando sessionsRouter en app.js

app.js

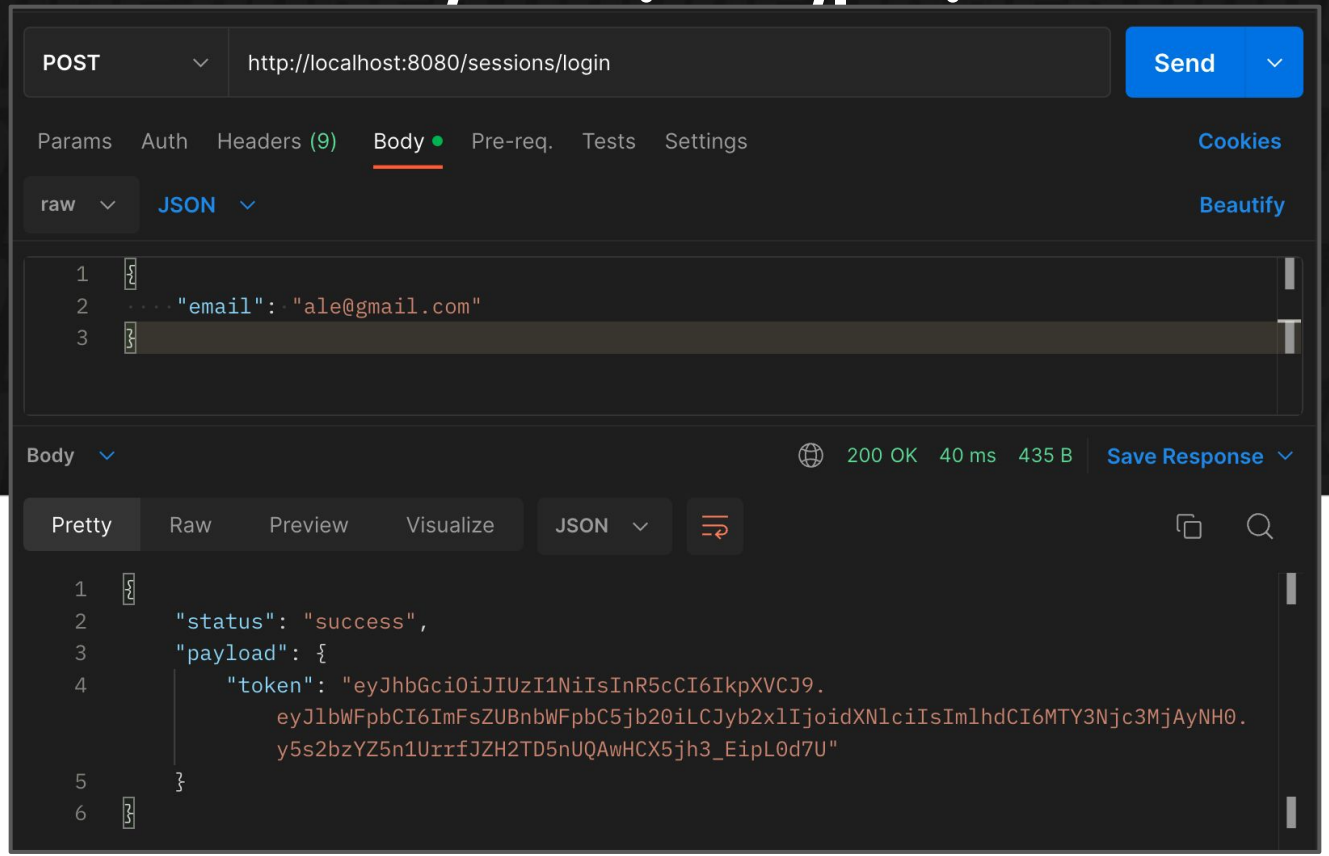
src > app.js > ...

```
1  import express from "express";
2  import UsersRouter from "./routes/users.router.js";
3  import SessionsRouter from "./routes/sessions.router.js";
4
5  const app = express();
6  app.use(express.json());
7
8  const usersRouter = new UsersRouter();
9  const sessionsRouter = new SessionsRouter();
10
11 app.use("/users", usersRouter.router);
12 app.use("/sessions", sessionsRouter.router);
13
14 app.listen(8080, () => {
15   console.log("Server listening on port 8080");
16 });
```

Probando en Postman la ruta /users/me SIN LOGUEARME

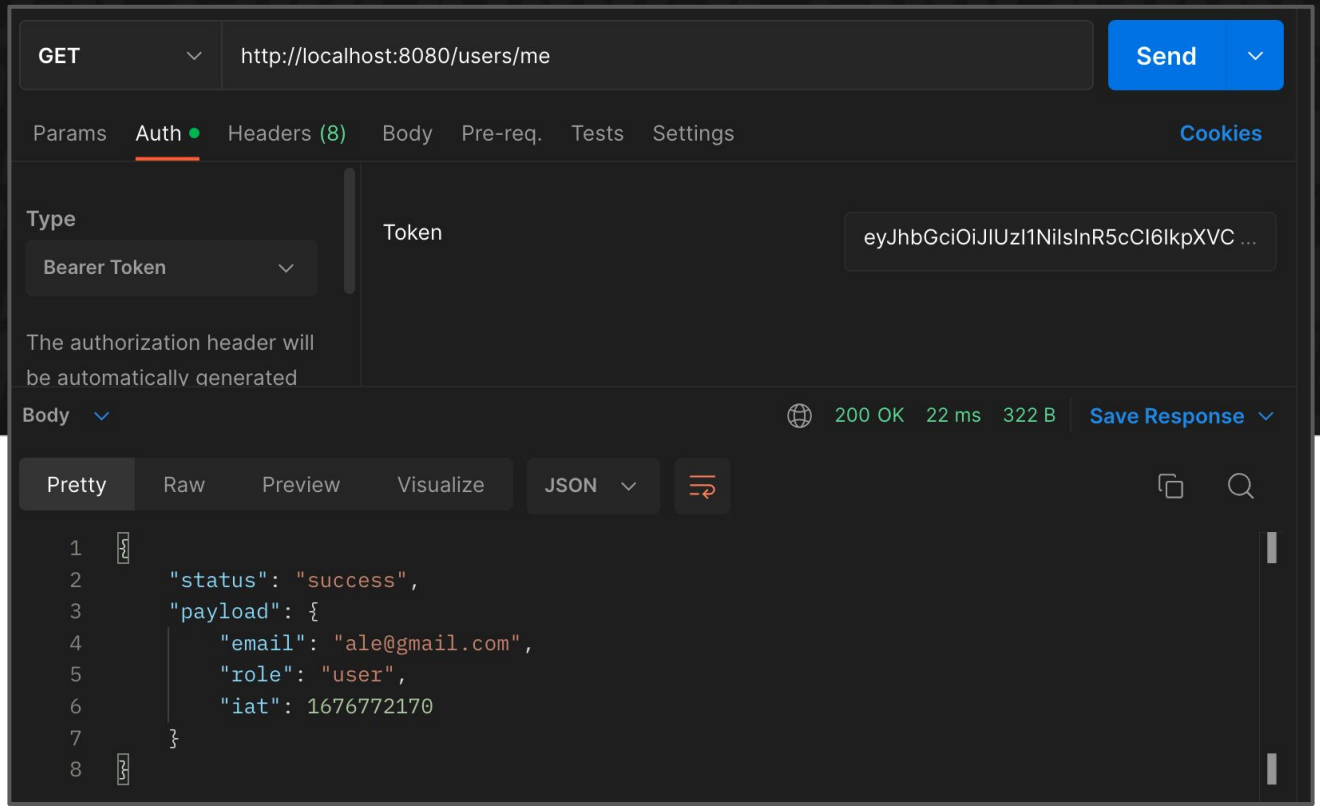


Obteniendo token a traves de



CODERHOUSE

Llamando a /users/me enviando el token (el rol coincidió con el arreglo de políticas)



CODERHOUSE

¡Atención!

La próxima clase será una **Práctica Integradora.** Te recomendamos prepararte con la guía de temas disponible en la carpeta de tu comisión

CODERHOUSE

¿Preguntas?

Resumen
de la clase hoy

- ✓ Estrategias avanzadas de router
- ✓ Creación de un Custom Router
- ✓ Manejo de sistemas de políticas.

Opina y valora esta clase

Muchas gracias.

#DemocratizandoLaEducación